Symbolic Simulation of Dataflow Synchronous Programs with Timers

Guillaume Baudart, Timothy Bourke, Marc Pouzet

Symbolic Simulation of Dataflow Synchronous Programs with Timers

Guillaume Baudart, Timothy Bourke, and Marc Pouzet

Abstract The synchronous language Lustre and its descendants have long been used to program and model discrete controllers. Recent work shows how to mix discrete and continuous elements in a Lustre-like language called Zélus. The resulting hybrid programs are deterministic and can be simulated with a numerical solver. In this article, we focus on a subset of hybrid programs where continuous behaviors are expressed using timers, nondeterministic guards, and invariants, as in Timed Safety Automata. We adapt a type system for mixing timers and discrete components and propose a source-to-source compilation pass to generate discrete code that, coupled with standard operations on Difference-Bound Matrices, produces symbolic traces that each represent a set of concrete traces.

Keywords: Symbolic Simulation; Synchronous Languages; Timed Automata; Hybrid Systems; Compilation; Type System

1 Introduction

Synchronous languages like Lustre [13] are ideal for programming an important class of embedded controllers. Their discrete model of time and deterministic semantics facilitate the precise expression of reactive behaviors. That said, many systems are naturally modeled using physical timing constraints that almost inevitably involve some 'timing nondeterminism' due to tolerances in requirements or uncertainties

Guillaume Baudart
IBM Research, e-mail: guillaume.baudart@ibm.com

Timothy Bourke
Inria Paris, École normale supérieure, PSL University, e-mail: timothy.bourke@inria.fr

Marc Pouzet
Sorbonne Universités UPMC Univ Paris 06, École normale supérieure, PSL University, Inria Paris, e-mail: marc.pouzet@ens.fr

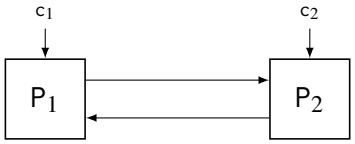


Fig. 1 A simple two-node application with clock inputs c_1 and c_2.

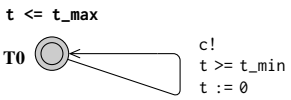


Fig. 2 A simple clock model with clock output c [30, Figure 4].

in implementations. Conversely, such constraints are readily modeled using Timed Automata [2, 19], and simulated symbolically in Uppaal [5, 23], but large-scale discrete-time behaviors are more cumbersome to express in such tools. In this article we try to have the best of both worlds by incorporating features of Timed Automata into a Lustre-like synchronous language. We focus on programming language design and symbolic simulation rather than verification.

As a simple running example, consider the application shown in Figure 1: two components named P_1 and P_2 are periodically triggered by local clocks c1 and c2. The clocks are subject to jitter and are modeled by the Timed Automaton shown in Figure 2. This model generates a signal c with a nominal period of $(T_{\max} + T_{\min})/2$ and a jitter of $\pm(T_{\max} - T_{\min})/2$. The timing constraints are expressed using a timer variable t that is reset to 0 at every emission of the signal c; the transition that emits c may occur whenever $t \geq T_{\min}$; and t must never exceed $T_{\max}$. Putting two or more such clocks in parallel and using their respective clock signals to trigger a program is a standard way to model communicating components [12, 30].

Classic synchronous languages are ideal for expressing the dynamic behavior of the components P_1 and P_2 but cannot easily express the continuous-time dynamics of the overall architecture. Zélus[1] [11] is a newer synchronous language that allows mixing both discrete and continuous elements. The timer t of Figure 2 can be simulated by a simple ordinary differential equation, $\dot{t} = 1$ and jitter can be introduced by resetting the timer to arbitrary values between $-T_{\min}$ and $-T_{\max}$ whenever t reaches 0. This approach, however, forces the programmer to make explicit implementation choices that are not part of the specification and it is not modular. Adding constraints on t, like another invariant, requires adapting its defining equation.

We propose to instead express *guards* and *invariants* directly as in the following program in a variation of Zélus that we call ZSy.

```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0 reset c() → 0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}
```

As in Zélus, the keyword hybrid declares a continuous-time component clock, parametrized by t_min and t_max, whose output c is defined by three concurrent equations. The first equation declares a timer—that is, a variable t where $\dot{t} = 1$—with initial value 0 and that is reset to 0 at each occurrence of c. We use the dedicated

[1] http://zelus.di.ens.fr

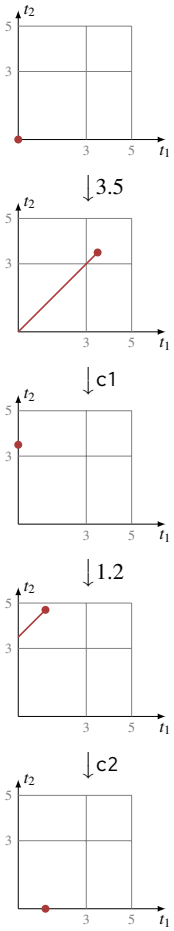


Fig. 3 A concrete simulation trace of `scheduler(3,5)`: t_1 and t_2 denote the values of the two timers, one for each quasi-periodic clock.

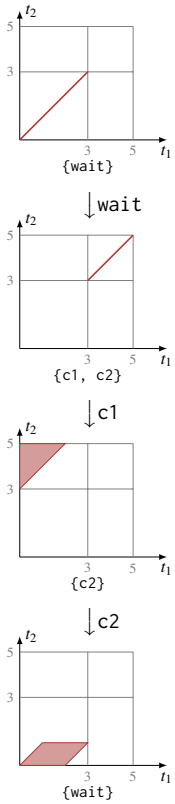


Fig. 4 A symbolic simulation trace of `scheduler(3,5)`. Each step corresponds to a set of timer values and a set of enabled transitions (below).

`timer` keyword to emphasize the focus on timed systems with limited continuous dynamics. The second equation states that the signal `c` *may* be emitted whenever $t \geq T_{\text{min}}$. The third equation declares an invariant stating that the value of t *must* never exceed T_{max}. We use braces to distinguish constraints from boolean conditions.

A model of a simple two-node architecture can be obtained by instantiating the `clock` function twice.

```
let hybrid scheduler(t_min, t_max) = c1, c2 where
  rec c1 = clock(t_min, t_max)
  and c2 = clock(t_min, t_max)
```

Signals produced by the `scheduler` function can then be used to trigger discrete controllers (like $\mathtt{P}_1$ and $\mathtt{P}_2$) written in the discrete subset of Zélus.

The traces of such systems comprise two kinds of events: time elapsing and discrete transitions triggered by signal emissions. Figure 3 shows a possible execution trace of the two-node architecture of Figure 1 with $T_{\text{min}} = 3$ and $T_{\text{max}} = 5$. Variables t_1 and t_2 denote the values of the two timers, one for each instance of `clock`. Starting from $t_1 = t_2 = 0$, `c1` is triggered when $t_1 = 3.5$. Then `c2` is triggered when $t_2 = 4.7$, that is, 1.2 after `c1`.

For the kind of systems we consider, that is, nondeterministic timed discrete-event systems, an execution is a sequence of discrete events (here, the clock ticks). Rather than simulating one concrete trace that assigns a precise date to each event, we employ an alternative simulation scheme that focuses on the ordering of events. Precise timer valuations are then replaced by symbolic sets called *zones* that encompass the timer valuations that give rise to the same sequences of discrete events.

At each step, the user or an external 'oracle' program chooses from a set of possible transitions. A transition means either waiting for a change in the set of enabled guards or firing enabled guards. If the `wait` transition is chosen, we compute the new zone by letting time elapse until the next change in the set of enabled guards as permitted by invariants in the program. Otherwise, firing guards triggers discrete-time computations, possibly resets some timers, and returns a new initial zone. The new zone is obtained by letting time elapse from this initial zone until the next change in the set of enabled guards.

The symbolic trace that contains the concrete simulation of Figure 3 is presented in Figure 4.

1. The simulation starts with $\{t_1 = t_2 = 0\}$, where no guards are enabled. The first zone is obtained from this initial position by letting time elapse until just before one or more guards become enabled, giving $\{0 \leq t_1 = t_2 < 3\}$. In this zone, the user has no choice but to `wait`.
2. The next zone is generated by letting time elapse as long as permitted by the invariants, giving $\{3 \leq t_1 = t_2 \leq 5\}$. In this zone, the user may choose `c1` or `c2` but not `wait`.
3. The user chooses `c1` which resets timer t_1 to 0. The new initial zone is then $\{t_1 = 0 \wedge 3 \leq t_2 \leq 5\}$, from which the new zone $\{3 \leq t_2 - t_1 \leq 5 \wedge 3 \leq t_2 \leq 5\}$ is obtained by letting time elapse within the limits imposed by invariants and guards, and from which only `c2` is possible.
4. The user chooses `c2` which resets timer t_2 to 0. The new initial zone is then $\{t_2 = 0 \wedge t_1 \leq 2\}$, from which the new zone obtained by letting time elapse is $\{0 \leq t1 - t2 \leq 2 \wedge t_1 < 3 \wedge t_2 < 1\}$, and from which no guards are enabled. The only possibility is to `wait` until the next change in the set of enabled guards.

This symbolic trace includes all concrete trace prefixes where `c1` occurs before `c2`. If a program reaches a state with contradictory constraints, or where no transitions are enabled, the simulation becomes stuck—such programs are deemed invalid.

The symbolic representation of timer valuations and zone-based simulation are standard practice for simulating and model checking Timed Automata models, no-

tably in the Uppaal tool [5, 23]. Our simulations differ in that we introduce explicit `wait` transitions between zones with differing sets of enabled transitions. This is natural in our setting where the idea is to 'steer' a synchronous program through time using additional inputs whose simultaneous occurrence is possible and whose absence is significant. In any case, the Uppaal simulation scheme is readily recovered.

Our main contributions are (1) to present a novel Lustre-like language that incorporates features of Timed Automata, (2) to adapt a type system that distinguishes discrete computations from continuous ones (3) to show how to compile programs to generate symbolic simulations using a novel source-to-source transformation, and (4) using a 'sweeping' construction to introduce explicit `wait` transitions.

The presentation is structured as follows. Section 2 recalls the standard data structure used to represent zones and introduces the new construction for `wait` transitions. The formal syntax of ZSy is presented in Section 3. We detail the type system in Section 4 and the source-to-source compilation pass in Section 5. In Section 6 we discuss how to extend ZSy with valued signals and automata. Related work is discussed in Section 7 before the conclusion in Section 8.

2 Difference-Bound Matrices

Difference-bound matrices [6,9,16][2] are a well-known data structure for representing and manipulating zones. DBMs are simple to implement and form a closed set with respect to both discrete transitions (mode changes, resets, intersections) and continuous evolution (time elapsing).

Let $\mathscr{T} = \{t_i\}_{0 \leq i \leq n}$ be a set of timer variables, with the convention that $t_0 = 0$. A DBM encodes a set of *difference constraints*, each of the form $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$, by gathering them into a $|\mathscr{T}| \times |\mathscr{T}|$ matrix of pairs of bounds and relations. Each timer variable is assigned a row and column: the row stores the upper bounds on the differences between the timer and all other timers and the column stores lower bounds. Figure 5 shows an example of a set of constraints and its representation as a DBM. We write $(d_{ij}, \preceq^d_{ij})_{0 \leq i,j \leq |\mathscr{T}|}$ to denote the coefficients of a DBM D.

2.1 Distances

The simulation scheme involves detecting changes in the set of enabled guards as time passes, that is, as the initial zone moves along the vector $(1, 1, \ldots, 1)$. Each guard G divides the state space of timer values into three zones: before activation, during activation, and after activation. A guard is enabled whenever the intersection between its activation zone and the current zone is not empty. Given an initial zone Z,

[2] We thank L. Fribourg for bringing the second reference to our attention.

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \le t_2 \\ 5 < t_3 \le 12 \\ 4 \le t_1 - t_2 \le 8 \end{array}\right\} \qquad \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} (0,\le) & (0,\le) & (-6,\le) & (-5,<) \\ (20,<) & (0,\le) & (8,\le) & (\infty,<) \\ (\infty,<) & (-4,\le) & (0,\le) & (\infty,<) \\ (12,\le) & (\infty,<) & (\infty,<) & (0,\le) \end{bmatrix} \end{array}$$

Fig. 5 Example of a set of constraints (left) and a corresponding DBM (right).

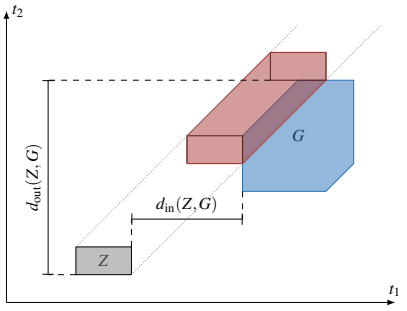


Fig. 6 Activation and deactivation distances of a guard G from an initial zone Z. The dotted line represents the zone obtained by letting time elapse indefinitely from Z. The red region is the zone where the guard is enabled.

we compute two distances for each guard G: $d_{\text{in}}(Z,G)$, the maximum distance before activation becomes possible, and $d_{\text{out}}(Z,G)$, the distance before deactivation. They are illustrated in Figure 6 (which is unrelated to the example in Figure 5).

A distance is a pair $(d,\prec)$ where $\prec$ specifies whether the limit is strict or not. Consider, for instance, a guard with activation zone $\{3 < t < 5\}$ and the initial zone $\{t = 0\}$. The distance before activation is $(3,<)$ which means that the guard is only enabled strictly after $t = 3$. The distance before deactivation is $(5,\le)$ since the guard is disabled when $t = 5$.

The distance before activation is obtained by comparing the upper bounds of the initial zone Z, with the lower bounds of the guard zone G (argmin is the index of the minimal value in a set):

$$d_{\text{in}}(Z,G) = (-g_{0j} - z_{j0}, \prec^g_{0j}) \text{ with } j = \underset{1\le i\le|\mathscr{T}|}{\text{argmin}}\{-g_{0i} - z_{i0}\}$$

The distance before deactivation is obtained by comparing the lower bounds of the initial zone with the upper bounds of the guard zone; $\overline{\prec}$ denotes the 'other' relation: $\overline{\le} = <$ and $\overline{<} = \le$.

$$d_{\text{out}}(Z,G) = (g_{j0} + z_{0j}, \overline{\prec^g_{j0}}) \text{ with } j = \underset{1\le i\le|\mathscr{T}|}{\text{argmin}}\{g_{i0} + z_{0i}\}$$

2.2 Sweeping

The sweeping operation generates the succession of zones in a symbolic simulation as time passes, that is, when the user chooses the `wait` transition. Each zone is

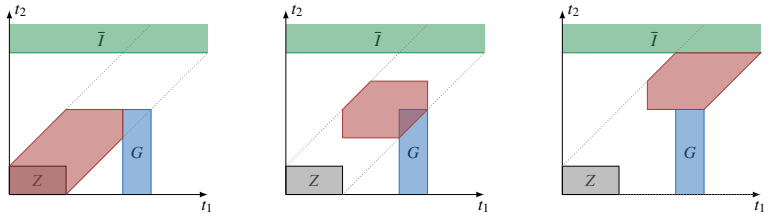


Fig. 7 Succession of zones from an arbitrary initial zone Z, guard G, and invariant I (the complement of I is shown to improved readability).

characterized by a set of enabled guards. The next zone is obtained by *sweeping* the initial zone along the vector $(1,1,\ldots,1)$ from its last position until the next change in the set of enabled guards. Figure 7 illustrates two successive `wait` transitions from the initial zone Z relative to a single guard G.

To compute the succession of zones from an initial zone Z, we compute the two distances associated with each guard that would become enabled if time were left to elapse indefinitely. We include the distance $(0,\leq)$ for the initial state, and $(\infty,\leq)$ for the final state when no more guards are reachable. These distances are ordered lexicographically, with $<$ less than $\leq$:

$$(n_1,\prec_1) < (n_2,\prec_2) \equiv n_1 < n_2 \text{ or } (n_1 = n_2 \text{ and } \prec_1 = <).$$

Each pair of successive distances defines a zone obtained by sweeping the initial zone on the corresponding interval. By construction, the succession of zones map the changes in the set of enabled guards as illustrated in Figure 7 (which is not directly related to the example of Figure 6).

The `zsweep` operation sweeps an initial zone Z through the interval defined by distances $(d_1,\prec_1)$ and $(d_2,\prec_2)$. The resulting zone C starts from the initial zone delayed by $(d_1,\prec_1)$, that is, we add d_1 to the lower bounds of Z and take $\prec_1$ as the limit relation (active after the change):

$$\forall 1 \leq i < |\mathscr{T}|,\ C_{0i} = (z_{0i} - d_1, \prec_1).$$

Zone C ends just before the next change, that is, after a distance $(d_2,\prec_2)$. We thus add d_2 to the upper bounds of Z and take $\overline{\prec_2}$ as the limit relation (stop before the next change):

$$\forall 1 \leq i < |\mathscr{T}|,\ C_{i0} = (z_{i0} + d_2, \overline{\prec_2}).$$

For the example with $Z = \{t = 0\}$ and $G = \{3 < t < 5\}$, the set of distances is $\{(0,\leq),(3,<),(5,\leq),(\infty,\leq)\}$, which gives three zones: $\{0 \leq t \leq 3\}$ where the guard is disabled, $\{3 < t < 5\}$ where the guard is enabled, and $\{5 \leq t < \infty\}$ where the guard is disabled again.

Invariants are accounted for by intersecting them with the results of the sweeping mechanism. For instance, adding the invariant `always {t ≤ 10}` to the example, gives the three zones: $\{0 \leq t \leq 3\}$, $\{3 < t < 5\}$, and $\{5 \leq t \leq 10\}$.

2.3 Zone interface

Our compilation and simulation routines require the small library of zone-manipulating operations summarized below.

`zall`	The complete space (unconstrained zone).
`zmake(c)`	Builds a DBM from a single constraint `c`.
`is_zempty(z)`	Returns *true* if DBM `z` denotes an empty zone.
`zreset(z,t,v)`	Resets a timer `t` to the value `v` in zone `z`.
`zinter(z1, z2)`	Returns the intersection of zones `z1` and `z2`.
`zinterfold(zv)`	Returns the intersection of a list of zones `zv` (fold `zinter` on `zv` starting from `zall`).
`zup(z)`	Lets time elapse indefinitely from zone `z` by removing all upper bounds.
`zenabled(zc, gv)`	Returns a list of booleans characterizing the set of enabled guards in the list `gv`. A guard is enabled if its activation zone $\texttt{gv}_i$ intersects the current zone `zc`.
`zdist(zi, g)`	Returns the activation and deactivation distances of a guard activation zone `g` from the initial zone `zi`.
`zdistmap(zi, gv)`	Returns the list of distances between an initial zone `zi` and a list of guard activation zones `gv`.
`zsweep(zi, d1, d2)`	Sweeps `zi` between distances `d1` and `d2`.

3 ZSy: a synchronous language with timers

We now present the kernel language of ZSy, a single assignment dataflow synchronous language extended with timers, invariants, and nondeterministic guards.

The formal syntax of ZSy is presented in Figure 8. A program is a sequence of *declarations* (d) of n-ary functions. As in Zélus, functions are declared as continuous-time (`hybrid`), discrete-time (`node`), or combinatorial. The *patterns* (p) in function arguments are nested pairs of variables. An *expression* (e) is either a variable x, a constant v, the application of an external operator op or a function f, a pair, or an initialized unit delay (e `fby` e),[3] and may refer to a set of locally recursive equations E. Several types of *equation sets* E are supported: simple equations where a variable is defined by an expression, parallel compositions (`and`), piecewise constant variables defined by a list of reset handlers and either an initial expression (`present`/`init`) or a default expression (`present`/`else`), timers with an initial value and a list of reset handlers (`timer`), invariant declarations (`always`) and signals that may be emitted when a constraint is satisfied (`emit`). Compared to Zélus, these last three equations are specific to ZSy. A list of *reset handlers* h comprises a sequence

[3] The declaration `x = e0 fby e` defines a stream x where $x(0) = [\![\texttt{e0}]\!](0)$ and for all $n > 0$, $x(n) = [\![\texttt{e}]\!](n-1)$.In other words, it takes its initial value from `e0` and thereafter is equal to `e` delayed by one instant.

$$
\begin{array}{lll}
d ::= \texttt{let hybrid } f(p) = e & E ::= x = e & h ::= e \to e \mid \cdots \mid e \to e \\
\quad \mid \texttt{let node } f(p) = e & \quad \mid E \texttt{ and } E & c ::= \Delta \sim e \mid c \texttt{ \&\& } c \\
\quad \mid \texttt{let } f(p) = e & \quad \mid x = \texttt{present } h \texttt{ init } e & \Delta ::= x \mid x - x \\
\quad \mid d\ d & \quad \mid x = \texttt{present } h \texttt{ else } e & \sim ::= < \mid \leq \mid \geq \mid > \\
p ::= x \mid (p,\ p) & \quad \mid \texttt{timer } x \texttt{ init } e \texttt{ reset } h & \\
e ::= x \mid v \mid op(e) \mid f(e) & \quad \mid \texttt{always \{ } c \texttt{ \}} & \\
\quad \mid (e,\ e) \mid e \texttt{ fby } e & \quad \mid \texttt{emit } x \texttt{ when \{ } c \texttt{ \}} & \\
\quad \mid e \texttt{ where rec } E & &
\end{array}
$$

Fig. 8 Syntax of ZSy.

of condition/expression pairs $x_1 \to e_1 \mid \cdots \mid x_n \to e_n$. Conditions must be boolean expressions in discrete-time contexts and signals in continuous-time contexts. When condition x_i is enabled (and $\forall j < i$, x_j is not enabled), the handler h takes the value of the corresponding expression e_i. When no condition is enabled, equations either keep their previous value (`init`) or take a default value (`else`). The form of *constraints* c is limited to allow for its encoding as a DBM.

For simplicity, the only possible action when a guard is fired is to emit a signal. This is not a real restriction since the `present` constructs allow signals to trigger arbitrary discrete-time computations. Since ZSy is a single assignment kernel, no signal can be emitted by two or more distinct guards.

4 Static typing

As in Zélus, we must statically discriminate between *discrete* and *continuous* computations. In ZSy, the transition between *continuous* and *discrete* contexts is realized via signals emitted by the guards. A variable is typed *discrete* if it is activated on signal emissions, and *continuous* otherwise. We can thus adapt the Zélus type system presented in [7, §3.2] to ZSy.

4.1 Types and kinds

Each function has a type of the form $t_1 \xrightarrow{k} t_2$ where k is a *kind* with three possible values: 'C' denotes continuous functions that can only be used in continuous contexts, 'D' denotes discrete functions that must be activated on the emission of a signal, and 'A' denotes a function that can be used in any context. The subkind relation $\subseteq$ is defined as $\forall k$, $k \subseteq k$ and $\mathrm{A} \subseteq k$. The type language is:

$$
\begin{array}{ll}
t & ::= t \times t \mid \alpha \mid bt \\
k & ::= \mathrm{D} \mid \mathrm{C} \mid \mathrm{A} \\
bt & ::= \texttt{int} \mid \texttt{bool} \mid \texttt{signal} \mid \texttt{timer} \\
\sigma & ::= \forall \alpha_1, \ldots, \alpha_n.\ t \xrightarrow{k} t
\end{array}
$$

A type (t) can be a pair ($t \times t$), a type variable (α) or a base type (bt). The base types are `int` and `bool` for constants, `signal` for signals emitted by guards, and `timer` for timer variables. Timers have a particular type to prevent their concrete values being used in an expression. Functions are associated to a type scheme σ where type variables are generalized.

A global environment G tracks the type schemes of functions, and another environment H assigns types to variables. We write $x : t$ to state that x is of type t, and if H_1 and H_2 are two environments, $H_1 + H_2$ denotes their union provided their domains are disjoint.

Generalization and instantiation:: Type schemes are obtained by generalizing the free variables in function types $t_1 \xrightarrow{k} t_2$:

$$gen(t_1 \xrightarrow{k} t_2) = \forall \alpha_1, \ldots, \alpha_n.\, t_1 \xrightarrow{k} t_2 \text{ where } \{\alpha_1, \ldots, \alpha_n\} = ftv(t_1 \xrightarrow{k} t_2),$$

where $ftv(t)$ denotes the set of free type variables in type t.

A type scheme can be instantiated by substituting type variables with actual types. $Inst(\sigma)$ denotes the set of possible instantiations of a type scheme σ. The kind of a type $t_1 \xrightarrow{k} t_2$ can be instantiated with any kind k' where $k \subseteq k'$:

$$\frac{k \subseteq k'}{(t \xrightarrow{k'} t')[t_1/\alpha_1, ..., t_n/\alpha_n] \in Inst(\forall \alpha_1, ..., \alpha_n.\, t \xrightarrow{k} t')}$$

4.2 Typing rules

Typing is defined by four judgments which resemble those of Zélus:

(TYP-EXP) $G, H \vdash_k e : t$

(TYP-ENV) $G, H \vdash_k E : H'$

(TYP-PAT) $\vdash_{pat} p : t, H$

(TYP-HANDLER) $G, H \vdash_k h : t$

The judgment (TYP-EXP) states that in environments G and H, expression e has kind k and type t. The judgment (TYP-ENV) states that in environments G and H, a set of equations E has kind k and produces a type environment H'. The judgment (TYP-PAT) states that a pattern p has type t and defines a type environment H. The judgment (TYP-HANDLER) states that in environments G and H, the value defined by a handler h has type t and kind k.

We add a fifth judgment:

(CHECK-ZONE) $G, H \vdash_{zone} c$

to check if a constraint defines a valid zone. In particular, (CHECK-ZONE) requires that the definition of zones only involve timer differences and integer bounds.

The initial environment G_0 contains the type of primitive operators, like `fby`, and imported operators, like (+) and (=).

$$\begin{aligned}(+) &: \texttt{int} \times \texttt{int} \xrightarrow{\mathrm{A}} \texttt{int}\\ (=) &: \forall\alpha,\ \alpha \times \alpha \xrightarrow{\mathrm{A}} \texttt{bool}\\ \texttt{fby} &: \forall\alpha,\ \alpha \times \alpha \xrightarrow{\mathrm{D}} \alpha\end{aligned}$$

Imported operators have kind A since they can be used in any context. The unit delay `fby` has kind D since it is only allowed in discrete contexts.

The typing rules are presented in Figure 9. An equation x `=` e is well typed if the types of x and e coincide. The kind of the equation is the kind of e. The parallel composition of two sets of equations E_1 `and` E_2 is well typed if both E_1 and E_2 are well typed and of the same kind. The equation x `=` `present` h `init` e_0 activates at instants defined by the handler h. The equation is well typed if the handler is well typed with the same type t as the initialization expression e_0. This equation may be used in the contexts allowed by the handler. The initialization value must have kind D even in continuous contexts. When a default value is provided it must have the same type and kind as the handler h. In particular, in continuous contexts the default value is not guarded by a signal and must thus have kind C.

The next three equations are specific to ZSy and can only be used in a continuous context. The equation `timer` x `init` e_0 `reset` h defines a variable of type `timer`. It is well typed if the reset handler h and initialization expression e_0 are well typed as `int`. The reset handler must have kind C and the overall kind is C. The equation `always { c }` introduces an invariant and does not define a variable. This equation is well typed if it has kind C and if the constraint c is a valid zone. The equation `emit` s `when { c }` is well typed if constraint c defines a valid zone. Variable s is then of type `signal` and the overall kind is C.

The typing of constants is illustrated with the integer constant 42. Constants can be used in any context. A variable of type t can be used in any context. A pair `(`e_1`,` e_2`)` has type $t_1 \times t_2$ if e_1 has type t_1 and e_2 has type t_2; e_1 and e_2 must have the same kind. An application f`(`e`)` has type t' if e has type t and if $t \xrightarrow{k} t'$ is a valid instantiation of the type scheme of f. The kind of the application $f(e)$ is given by the kind of f. A local definition e `where rec` E is well typed if the set of equations E is well typed and expression e is well typed in the extended environment.

A function definition has type $t_1 \xrightarrow{k} t_2$ if the input pattern p has type t_1 and the defining expression has type t_2. Function types are generalized. The kind is given in the definition: `hybrid` for C, `node` for D, and nothing for kind A. Function definitions are typed sequentially. Typing patterns yields an environment containing the types of the variables within. A handler c_1 `->` e_1 `|` $\cdots$ `|` c_n `->` e_n is well typed if all expressions e_i have the same type t and conditions c_i have type `signal` in continuous contexts or `bool` in discrete contexts. The expressions e_i must have kind D since, in any case, they are only activated at discrete instants. Finally, a constraint c defines a

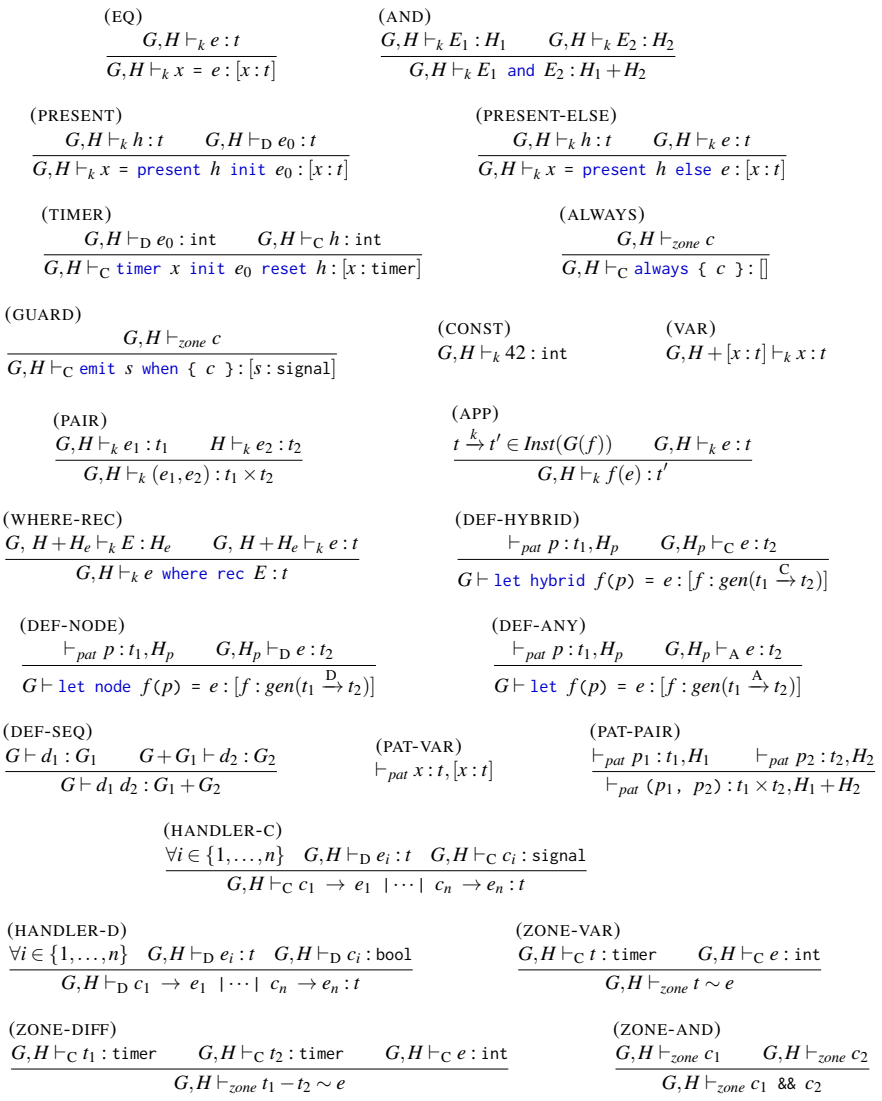


Fig. 9 The typing rules.

valid zone if variables are of type `timer` and bounds are of kind C and type `int`; the values are thus piecewise constant and can only change on signal emissions.

Since expressions of kind A can be executed in any context we also have the following subtyping property by induction on the typing derivation of $G,H \vdash_{\mathrm{A}} e : t$.

Property 1 (Subtyping). $G,H \vdash_{\mathrm{A}} e : t \implies (G,H \vdash_{\mathrm{C}} e : t) \wedge (G,H \vdash_{\mathrm{D}} e : t)$.

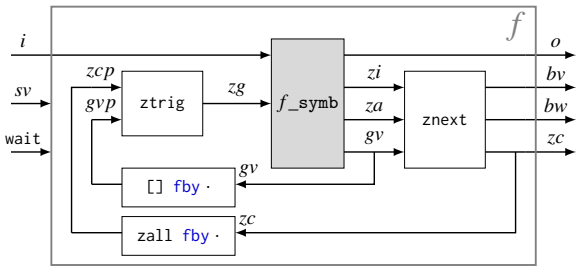


Fig. 10 Compiling a top-level continuous-time function `let hybrid` $f(i)$ `=` o into a discrete-time one `let node` $f(\texttt{wait}, sv, i)$ `=` o, bv, bw, zc for simulation. The function f`_symb` results from the source-to-source transformation of f.

5 Compilation

ZSy programs are compiled modularly for symbolic simulation by rewriting the continuous-time components, that is, those marked `hybrid`, into the purely discrete-time subset of the language. The resulting programs have additional inputs and outputs to handle zones and nondeterministic choices. They manipulate abstract DBM values using the small set of operations introduced in Section 2. The compilation produces valid discrete Zélus code that can be compiled by the Zélus toolchain.

At each step of the simulation scheme described in Section 1, a user chooses either to `wait` or to fire enabled guards.

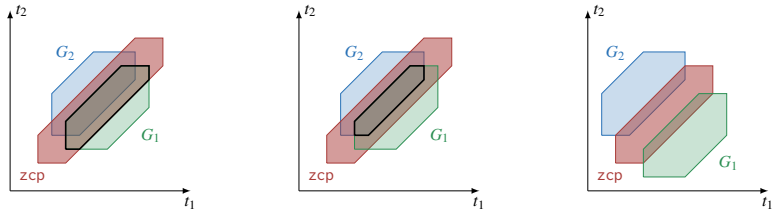
1. If the chosen transition is `wait` we compute the new zone by letting time elapse until the next change in the set of enabled guards if allowed by the active invariants.
2. Otherwise, firing guards triggers discrete-time computations, possibly resets timers, and returns a new initial zone obtained by letting time elapse from the current initial zone until the next change in the set of enabled guards.

Compiling the top-level function `let hybrid` $f(i)$ `=` o gives the result depicted in Figure 10. There are two additional inputs: *sv* a boolean vector where elements are set to *true* to fire guards, and `wait` a boolean set to *true* to fire the `wait` transition. And three additional outputs: *bv* a boolean vector characterizing the set of enabled guards, *bw* a boolean set to *true* if the `wait` transition is enabled, and *zc* the current zone.

The `wait` transition and the guard activations are mutually exclusive: if the input `wait` is set to *true*, other inputs are ignored. Unlike in Timed Automaton models, however, it is possible to simultaneously fire multiple guards. Indeed this is a typical feature of synchronous languages.

The generated function defines three execution phases:

1. Given the current zone *zcp* and the vector of guard activation zones *gvp* computed at the previous step, function `ztrig` computes the trigger zone *zg*.



Firing guard G_1 | Firing G_1 and G_2 | Exclusive guards

Fig. 11 Computing the trigger zone from the current zone `zcp`.

2. Function f`_symb` triggers the discrete-time computations and returns the initial zones *zi* obtained by applying the resets to *zg*, the conjunction of active invariants *za*, and the new vector of guard activation zones *gv*.
3. Function `znext` computes the new zone *zc* by letting time elapse from *zi* until the set of enabled guards changes.

5.1 Computing the trigger zone: `ztrig`

The function `ztrig` computes the trigger zone of the guards fired by the user by calculating their intersection with the current zone `zcp`.

```
let node ztrig(sv, zcp, gvp) = zg where
  rec fv = filter(gvp, sv)
  and zg = zinter(zcp, zinterfold(fv))
```

The list of the activation zones of the fired guards `fv` is obtained by filtering `gvp` according to `sv`, whose *true* elements indicate which guards to fire. Multiple guards may fire simultaneously, as illustrated in Figure 11. Firing mutually exclusive guards gives an empty zone and a blocked simulation. An alternative would be to detect this situation, signal it through an additional output, and remain in the same state.[4]

5.2 Source-to-source generation of f`_symb`

We adapt the source-to-source compilation scheme originally developed to compile Zélus programs for simulation with a numerical solver [7, §4]. The translation replaces timers, invariants, and guard definitions. Hybrid declarations in the source program `let hybrid` $f(i)$ `=` o become discrete-time declarations in the resulting program,

$$\texttt{let node } f\texttt{\_symb}(tv, \texttt{wait}, sv, zg, i) \texttt{ = } o, zi, za, gv.$$

[4] We thank R. von Hanxleden for his questions which led to this idea.

There are four new inputs: *tv*, a vector of timer identifiers; `wait`, a boolean for indicating a wait transition; *sv*, a boolean vector indicating the guards to fire; and *zg*, a trigger zone computed by `ztrig`. And three new outputs: *zi*, an initial zone; *za*, the conjunction of all invariants; and *gv*, a vector of guard activation zones. At the end of the compilation process, when combining f`_symb` with `ztrig` and `znext`, timer identifiers are replaced by unique values $1,2,3,\ldots,n$: the assigned dimensions in the manipulated DBMs. This approach permits the discrimination of timers defined in multiple instantiations of the same node (as in `scheduler`, for example).

The generation of f`_symb` is defined by the five mutually recursive functions presented in Figure 12:

TraDef(d) translates declarations. Only continuous-time declarations introduced by `hybrid` are modified.
Tra(zi,e) translates expressions using a variable *zi* to pass the currently computed version of the initial zone.
TraEq(zi,E) translates equations.
TraZ(zi,c) translates constraints.
TraH(zi,h) translates handlers.

In addition to the resulting expression, constraint, or handler, the last four functions return an intermediate result in the calculation of the new initial zone (*zi*), and five vector variables to accumulate invariant zones (*av*), guard activation zones (*gv*), new signal inputs (*sv*), new timer inputs (*tv*), and new equations (*E*).[5]

We now describe each of the definitions in Figure 12 from the top down.

As described above, the translation of a continuous-time function definition yields a new function with extra inputs and outputs. The translation of the defining expression gives a tuple $\langle e', zi', av, gv, sv, tv, E\rangle$. The initial zone computed incrementally from *zg* by the translated expression, zi', is only returned when a `wait` transition is not requested, otherwise the previous value of *zi* is returned, or `zall` initially, to be manipulated by the sweeping algorithm in `znext` (see Figure 10). The conjunction of the invariants in *av* is calculated with `zinterfold` and returned as *za*. The current guard activation zones are returned in *gv*. The vectors *sv* and *tv* are bound as inputs. Finally, any generated equations *E* are added to the node body.

Variables (x) and constants (v) are passed unchanged. They do not have any effect on the calculation of the initial zone, nor introduce any new zones, inputs, or equations. Neither do combinatorial function applications, external operators, or the tuple constructor: the results of a recursive call to *Tra*(zi,e) are simply transmitted. There are no rules for discrete-time function applications or `fby`s as the type system only permits such elements in discrete-time contexts—which are not transformed—and thus not directly within the body of a continuous-time function.

A continuous-time function application f`(`e`)` is replaced by a discrete-time one (r, zi_f, za, g) `=` f`_symb`$(t, wait, s, zi', e')$, introduced as a new equation. The fresh variables t and s pass the required timer identifiers and guard inputs, e' is the translation

[5] We write [] to denote the empty vector and the empty set of equations; $[x_1,\ldots x_n]$ @ $[y_1,\ldots,y_n] = [x_1,\ldots,x_n,y_1,\ldots,y_n]$ to denote the concatenation of two vectors; and $x_0 :: [x_1,\ldots,x_n] = [x_0,x_1,\ldots,x_n]$ to denote the addition of an element at the beginning of a vector.

$$
\begin{aligned}
&\mathit{TraDef}(\texttt{let hybrid } f\texttt{(}p\texttt{) = } e) && = \mathit{let}\ \langle e', zi', av, gv, sv, tv, E\rangle = \mathit{Tra}(zg, e)\ \mathit{in}\\
&&& \quad \texttt{let node } f\texttt{\_symb(}tv, wait, sv, zg, p\texttt{) = } e', zi, za, gv \texttt{ where}\\
&&& \quad\ \texttt{rec } E \texttt{ and } za \texttt{ = zinterfold(}av\texttt{)}\\
&&& \quad\ \texttt{and } zi \texttt{ = if } wait \texttt{ then (zall fby } zi\texttt{) else } zi'\\
&&& \quad \text{where } zi, \text{ and } za \text{ are fresh variables.}\\
&\mathit{Tra}(zi, x) && = \langle x, zi, [], [], [], [], []\rangle \qquad \text{Constants } (v) \text{ are treated likewise.}\\
&\mathit{Tra}(zi, op(e)) && = \mathit{let}\ \langle e', zi', av, gv, sv, tv, E\rangle = \mathit{Tra}(zi, e)\ \mathit{in}\\
&&& \quad \langle op(e'), zi', av, gv, sv, tv, E\rangle\\
&\mathit{Tra}(zi, (e_1,\ e_2)) && = \mathit{let}\ \langle e'_1, zi_1, av_1, gv_1, sv_1, tv_1, E_1\rangle = \mathit{Tra}(zi, e_1)\ \mathit{in}\\
&&& \quad \mathit{let}\ \langle e'_2, zi_2, av_2, gv_2, sv_2, tv_2, E_2\rangle = \mathit{Tra}(zi_1, e_2)\ \mathit{in}\\
&&& \quad \langle (e'_1,\ e'_2), zi_2, av_1 \mathbin{@} av_2, gv_1 \mathbin{@} gv_2, sv_1 \mathbin{@} sv_2, tv_1 \mathbin{@} tv_2,\\
&&& \quad\ E_1 \texttt{ and } E_2\rangle\\
&\mathit{Tra}(zi, f(e)) && = \mathit{let}\ \langle e', zi', av, gv, sv, tv, E\rangle = \mathit{Tra}(zi, e)\ \mathit{in}\\
&\quad \text{if } f \text{ is combinatorial} && \quad \langle f(e'), zi', av, gv, sv, tv, E\rangle\\
&\mathit{Tra}(zi, f(e)) && = \mathit{let}\ \langle e', zi', av, gv, sv, tv, E\rangle = \mathit{Tra}(zi, e)\ \mathit{in}\\
&\quad \text{if } f \text{ is continuous-time} && \quad \langle r, zi_f, za :: av, g :: gv, s :: sv, t :: tv,\\
&&& \quad\ E \texttt{ and } (r, zi_f, za, g) \texttt{ = } f\texttt{\_symb(}t, wait, s, zi', e'\texttt{)}\rangle\\
&&& \quad \text{where } r, zi_f, za, g, s, \text{ and } t \text{ are fresh variables.}\\
&\mathit{Tra}(zi, e \texttt{ where rec } E) && = \mathit{let}\ \langle zi_1, av_1, gv_1, sv_1, tv_1, E_1\rangle = \mathit{TraEq}(zi, E)\ \mathit{in}\\
&&& \quad \mathit{let}\ \langle e', zi_2, av_2, gv_2, sv_2, tv_2, E_2\rangle = \mathit{Tra}(zi_1, e)\ \mathit{in}\\
&&& \quad \langle e', zi_2, av_1 \mathbin{@} av_2, gv_1 \mathbin{@} gv_2, sv_1 \mathbin{@} sv_2, tv_1 \mathbin{@} tv_2, E_1 \texttt{ and } E_2\rangle\\
&&& \quad \text{assuming unique names for variables in } E.\\
&\mathit{TraEq}(zi, x \texttt{ = } e) && = \mathit{let}\ \langle e', zi', av, gv, sv, tv, E\rangle = \mathit{Tra}(zi, e)\ \mathit{in}\\
&&& \quad \langle zi', av, gv, sv, tv, E \texttt{ and } x \texttt{ = } e'\rangle\\
&\mathit{TraEq}(zi, E_1 \texttt{ and } E_2) && = \mathit{let}\ \langle zi_1, av_1, gv_1, sv_1, tv_1, E'_1\rangle = \mathit{TraEq}(zi, E_1)\ \mathit{in}\\
&&& \quad \mathit{let}\ \langle zi_2, av_2, gv_2, sv_2, tv_2, E'_2\rangle = \mathit{TraEq}(zi_1, E_2)\ \mathit{in}\\
&&& \quad \langle zi_2, av_1 \mathbin{@} av_2, gv_1 \mathbin{@} gv_2, sv_1 \mathbin{@} sv_2, tv_1 \mathbin{@} tv_2, E'_1 \texttt{ and } E'_2\rangle\\
&\mathit{TraEq}(zi, x \texttt{ = present } h \texttt{ init } e_0) && = \mathit{let}\ \langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h\rangle = \mathit{TraH}(zi, h)\ \mathit{in}\\
&&& \quad \langle zi_h, av_h, gv_h, sv_h, tv_h, E_h \texttt{ and } x \texttt{ = present } h' \texttt{ init } e_0\rangle\\
&\mathit{TraEq}(zi, x \texttt{ = present } h \texttt{ else } e) && = \mathit{let}\ \langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h\rangle = \mathit{TraH}(zi, h)\ \mathit{in}\\
&&& \quad \mathit{let}\ \langle e', zi', av, gv, sv, tv, E\rangle = \mathit{Tra}(zi_h, e)\ \mathit{in}\\
&&& \quad \langle zi', av_h \mathbin{@} av, gv_h \mathbin{@} gv, sv_h \mathbin{@} sv, tv_h \mathbin{@} tv,\\
&&& \quad\ E_h \texttt{ and } E \texttt{ and } x \texttt{ = present } h' \texttt{ else } e'\rangle\\
&\mathit{TraEq}(zi, \texttt{timer } t \texttt{ init } e_0 \texttt{ reset } h) && = \mathit{let}\ \langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h\rangle = \mathit{TraH}(zi, h)\ \mathit{in}\\
&&& \quad \mathit{let}\ x'_1 \to e_1 \mid \cdots \mid x'_n \to e_n = h'\ \mathit{in}\\
&&& \quad \langle zi_t, av_h, gv_h, sv_h, t :: tv_h,\\
&&& \quad\ E_h \texttt{ and } zi_t \texttt{ = present (true fby false)} \to \texttt{zreset(}zi_h, t, e_0\texttt{)}\\
&&& \qquad\qquad \texttt{| } x'_1 \to \texttt{zreset(}zi_h, t, e_1\texttt{) | } \cdots\\
&&& \qquad\qquad \texttt{| } x'_n \to \texttt{zreset(}zi_h, t, e_n\texttt{) else } zi_h\rangle\\
&&& \quad \text{where } zi_t \text{ is a fresh variable.}\\
&\mathit{TraEq}(zi, \texttt{always \{ } c \texttt{ \}}) && = \mathit{let}\ \langle c', zi_c, av_c, gv_c, sv_c, tv_c, E_c\rangle = \mathit{TraZ}(zi, c)\ \mathit{in}\\
&&& \quad \langle zi_c, za :: av_c, gv_c, sv_c, tv_c, E_c \texttt{ and } za \texttt{ = zmake(}c'\texttt{)}\rangle\\
&&& \quad \text{where } za \text{ is a fresh variable.}\\
&\mathit{TraEq}(zi, \texttt{emit } s \texttt{ when \{ } c \texttt{ \}}) && = \mathit{let}\ \langle c', zi_c, av_c, gv_c, sv_c, tv_c, E_c\rangle = \mathit{TraZ}(zi, c)\ \mathit{in}\\
&&& \quad \langle zi_c, av_c, zs :: gv_c, s :: sv_c, tv_c, E_c \texttt{ and } zs \texttt{ = zmake(}c'\texttt{)}\rangle\\
&&& \quad \text{where } zs \text{ is a fresh variable.}\\
&\mathit{TraH}(zi, x_1 \to e_1 \mid \cdots \mid x_n \to e_n) && = \mathit{let}\ \langle x'_i, zi_i, av_i, gv_i, sv_i, tv_i, E_i\rangle = \mathit{Tra}(zi_{i-1}, c_i)\ \mathit{in}\\
&&& \quad \langle x'_1 \to e_1 \mid \cdots \mid x'_n \to e_n, zi_n, av_1 \cdots \mathbin{@} av_n, gv_1 \cdots \mathbin{@} gv_n,\\
&&& \quad\ sv_1 \cdots \mathbin{@} sv_n, tv_1 \cdots \mathbin{@} tv_n, E_1 \cdots \texttt{ and } E_n\rangle\\
&&& \quad \text{where } zi_0 = zi.\\
&\mathit{TraZ}(zi, \Delta_1 \sim_1 e_1 \cdots \texttt{ \&\& } \Delta_n \sim_n e_n) && = \mathit{let}\ \langle e'_i, zi_i, av_i, gv_i, sv_i, tv_i, E_i\rangle = \mathit{Tra}(zi_{i-1}, e_i)\ \mathit{in}\\
&&& \quad \langle \Delta_1 \sim_1 e'_1 \cdots \texttt{ \&\& } \Delta_n \sim_n e'_n, zi_n, av_1 \cdots \mathbin{@} av_n, gv_1 \cdots \mathbin{@} gv_n,\\
&&& \quad\ sv_1 \cdots \mathbin{@} sv_n, tv_1 \cdots \mathbin{@} tv_n, E_1 \cdots \texttt{ and } E_n\rangle\\
&&& \quad \text{where } zi_0 = zi.
\end{aligned}
$$

Fig. 12 The source-to-source generation of f`_symb`.

of e, and zi' passes the initial zone calculated by the translated elements. The t and s variables, and those for the returned conjunction of invariants (za), and the vector of guard activation zones are added to the appropriate accumulators. The original outputs are returned in a fresh variable r, as is the updated initial zone, zi_f. The structure of nested function calls is reflected in the tree structure of sv, tv, and gv. Local definitions (e `where rec` E) are flattened which is sound provided that variables have unique identifiers. There are no side effects and equations can be reordered.

For simple equations (`x = e`), parallel compositions (`and`), and `present` constructs, we apply the translation recursively to compute the initial zone and accumulate inputs, guard activation zones, and invariants. Initial zone calculations are threaded from one side of a parallel composition to the other, which may, regrettably, introduce artificial causality constraints. These can be minimized by sorting the equations prior to their transformation. Otherwise, the Zélus compiler will inline function applications to break causality constraints if necessary. In the `present` construct, the initial expression e_0 and the handler bodies are discrete and do not require translation.

The translation of a timer definition adds a new timer identifier t to tv and an equation defining the updated initial zone zi_t by a `present`/`else` construct. A new handler ensures that the tth dimension of the initial zone is initially reset to the value of e_0, the other handlers are translated from the original construct with calls to `zreset` introduced to update the initial zone when necessary, otherwise the `else` branch passes the initial zone without modifying it.

Invariants are translated into an equation defined by `zmake`, which generates a DBM from the constraint expression.

Translating a signal emission adds an activation zone to gv and an element to the vector sv of user-controlled signals. The activation zone is defined by an equation. ZSy is a single assignment kernel, so a signal is only ever defined by a single guard.

5.3 Computing the current zone: `znext`

The discrete-time function `znext` computes the current zone zc and the set of enabled transitions using the information computed by f`_symb`, namely the initial zone `zi`, the conjunction of all invariants `za`, and the vector of guard activation zones `gv`.

```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(dl, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc ≠ zm)
```

When the user chooses to `wait`, the current zone `zc` is obtained via the sweeping mechanism described in Section 2.2. Sweeping restarts from `dzero` $= (0, \leq)$ whenever the user fires guards (otherwise f`_symb`'s outputs are unchanged).

We compute `dl`, the list of distances for reachable guards, from `zi` and `gv`. The function `mindist(dl, dp)` gives the smallest distance greater than `dp`. It enumerates the distance pairs in order; `dp = dzero fby d` is the distance reached at the previous step. Each successive distance pair `(dp, d)` defines a zone `zn` obtained by sweeping `zi` between `dp` and `d`. The current zone `zc` is the intersection of `zn` and the invariant `za`.

We compute `bv`, the vector characterizing the enabled guards, from `zc` and `gv`. The maximal zone `zm` is obtained from `zn` by letting time elapse indefinitely with the same invariant `za`. A `wait` transition is possible as long as `zc` $\neq$ `zm` (`bw`).

Returning `zm` rather than `zc` gives the Uppaal simulation scheme where a user may choose from all reachable guards.

5.4 Example

Figure 13 presents the result of the compilation of the `clock` and `scheduler` nodes defined in Section 1. Each continuous-time function is compiled into two discrete-time functions: the result of the source-to-source generation (e.g., `clock_symb` and `scheduler_symb`), and the wrapper presented in Figure 10. Note that the compilation is modular: `scheduler_symb` makes two calls to `clock_symb`. The `scheduler` function takes as additional inputs the signals `wait` and `(c1, c2)` which allows the user to control the symbolic simulation. Discrete-time functions are not modified. It is, for instance, possible to simulate the application shown in Figure 1 by activating the discrete controllers P_1 and P_2 on the signals `c1'` and `c2'` emitted by `scheduler`.

6 Extensions

In this section we discuss two possible extensions of ZSy to improve its expressiveness: valued signals and automata.

6.1 Valued signals

In ZSy, signals cannot carry values but it is relatively simple to add this feature to the language by reusing Zélus signals. The type of a signal α *signal* is parametrized by α, the type of its values. A pure signal, without any value, has type *unit signal*. An expression calculating a value to emit on a signal must be of kind D since emissions are discrete computations.

```
let node clock_symb(t, wait, c, zg, (t_min, t_max))
    = c, zi, za, [zs] where
  rec zit = present (true fby false) → zreset(zg, t, 0)
            | c → zreset(zg, t, 0)
            else zg
  and zs = zmake({t ≥ t_min})
  and zb = zmake({t ≤ t_max})
  and za = zinterfold([zb])
  and zi = if wait then (zall fby zi) else zit

let node clock(wait, c, (t_min, t_max)) = c', bv, bw, zc where
  rec zg = ztrig([c], zcp, gvp)
  and c', zi, za, gv = clock_symb(1, wait, c, zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv

let node scheduler_symb((t1, t2), wait, (c1, c2), zg, (t_min, t_max))
    = (c1', c2'), zi, za, gv1 @ gv2 where
  rec c1', zi1, za1, gv1 = clock_symb(t1, wait, c1, zg, (t_min, t_max))
  and c2', zi2, za2, gv2 = clock_symb(t2, wait, c2, zi1, (t_min, t_max))
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zi2

let node scheduler(wait, (c1, c2), (t_min, t_max))
    = (c1', c2'), bv, bw, zc where
  rec zg = ztrig([c1; c2], zcp, gvp)
  and (c1', c2'), zi, za, gv =
    scheduler_symb((1, 2), wait, (c1, c2), zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv
```

Fig. 13 Compilation of `clock` and `scheduler`.

We keep the syntax of Zélus for valued signals:

(*emission*) `emit` s [`=` e]
(*reception*) `present` $s(v)$ [`on` $P(v)$] → e

with the particular case `present` $s()$ → e for pure signals. The optional condition [`on` $P(v)$] allows to filter the value v received on a signal with a boolean predicate P directly in the branches of the `present` handler.

6.2 Automata

A major restriction of ZSy is the absence of state in continuous functions. Conditionals like `if` e_0 `then` e_1 `else` e_2 can be added as an external operator of arity 3, but in that case, the three expressions e_0, e_1, and e_2, and the equations produced by their

translations, are computed at every step. It is, however, possible to extend ZSy with hierarchical automata following the compilation technique introduced in [8].

```
let hybrid auto() = o where
  rec automaton
      | S1 → do o = 1
              and timer t1 init 0 reset c1 → 0
              and emit c1 when {t1 > 3}
              and always {t1 ≤ 5}
          until c1 then S2
      | S2 → do o = 2
              and timer t2 init 0 reset c2 → 0
              and emit c2 when {t2 > 2}
              and always {t2 ≤ 7}
          until c2 then S1
```

Automata in continuous contexts are translated into discrete automata where the signals triggering transitions between states are replaced by boolean conditions.

The easiest solution is to duplicate all equations introduced by timers, guards, and invariants during the translation such that if a variable is defined in one state of the automaton, the same variable returns a dummy value in all other states. The result of compiling the previous example is shown below.

```
let node auto_symb((t1, t2), wait, (c1, c2), zg) = o, zi, za, [zs1; zs2] where
  rec automaton
      | S1 → do o = 1
              and zi1 = present (true fby false) → zreset(zg, t1, 0)
                        | c1 → reset(zg, t1, 0)
                        else zg
              and zs1 = zmake({t1 > 3})
              and za1 = zmake({t1 ≤ 5})
              and zi2 = zall and zs2 = zempty and za2 = zall
          until c1 then S2
      | S2 → do o = 2
              and zi2 = present (true fby false) → zreset(zg, t2, 0)
                        | c2 → reset(zg, t2, 0)
                        else zg
              and zs2 = zmake({t2 > 2})
              and za2 = zmake({t2 ≤ 7})
              and zi1 = zall and zs1 = zempty and za1 = zall
          until c2 then S1
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zinterfold([zi1; zi2])
```

Each state of the automaton generates a possible initial zone *zi*. This variable takes the dummy value `zall` in all other states. We gather all these zones into a vector and the global initial zone is the intersection of its elements. The activation zone of a guard defined in one state is empty in all other states (the guard cannot be enabled). An invariant defined in one state becomes `zall` in all other states.

Following [8], it is also possible to minimize memory allocations by reusing variables across multiple states. For instance, the pairs of variables (`zi1`,`zi2`)

and $(\texttt{za1}, \texttt{za2})$ could be merged since they are used in exclusive states. However, we still need to gather all timer identifiers, guard signals, and guard activation zones for interaction with the user.

7 Related work

Our proposal is based on Timed Safety Automata [2, 19], and greatly influenced by the Uppaal[6] [5, 23] simulator where users choose among enabled transitions to navigate the reachable symbolic states of a model. Our approach differs in two main ways. First, we include explicit 'wait transitions' and develop algorithms for them (Sections 2.1 and 2.2). Second, Uppaal models are networks of Timed Automata with a C-like language for manipulating values, whereas we adapt techniques developed for synchronous languages that allow state-based behaviors to be encapsulated in named nodes and arbitrarily composed in parallel and hierarchically. Synchronous languages have efficient, modular compilation schemes and are regularly used to program large-scale embedded controllers. We treat simultaneous signal emission, integrate a rich language for discrete controllers and techniques for its efficient compilation, and give a meaning to timing constructs by translating them into a simpler discrete language. Compared to Zélus, we allow a limited form of nondeterminism in guard expressions and present a novel source-to-source compilation pass to simulate (some programs) symbolically rather than numerically.

Much has been written about verifying hybrid systems [1], typically using symbolic representations to over-approximate the states and traces of hybrid automata. This article does not address verification, and focuses instead on language design and compilation. Our symbolic representations are exact.

Our proposal may, nevertheless, be useful for semi-symbolic reachability analysis where discrete states are represented explicitly and sets of clock valuations are represented symbolically as in the Kronos [32], Uppaal [5, 23], and Red [31] model checkers. Programs in our 'extended version of Lustre' could potentially be compiled to C code and linked with the highly-tuned Uppaal DBM library and routines for efficiently representing and exploring discrete states. The state of the art, however, in verifying Lustre programs is the Kind 2 model checker [15], which is based on fully-symbolic SMT techniques. Such techniques can be adapted to treat timed automata [20, 22, 27], and potentially used to analyze programs in our language. The compilation scheme we propose is unnecessary in such an approach.

Several works propose adding nondeterministic constraints to synchronous languages. Lutin [28, 29] and the commercial Argosim Stimulus tool[7] are designed for testing discrete-time reactive programs. Inputs are specified nondeterministically and characterized at each simulation step by a set of possible values. Concrete rather than symbolic traces are generated by choosing values randomly. Yo-yo [17, 25] is a tool

[6] `http://www.uppaal.org`

[7] `http://argosim.com`

for the symbolic simulation of discrete-time dataflow programs extended with nondeterministic relations between variables. The simulator generates symbolic states that represent sets of possible variable values. Rather than add nondeterminism to variable values, we add timing nondeterminism to continuous-time constructs—variables still evolve deterministically. Compilation still produces a (discrete) synchronous program, but the discretization is chosen based on the timing constraints.

There have been several propositions for integrating continuous time into synchronous programs, usually for verifying real-time properties. The semantics of Argos programs extended with timeouts and watchdogs [21] was expressed as timed automata for verification in the Kronos model checker [32]. Similarly, Taxys [10] allows the annotation of Esterel programs with real-time characteristics and verifies them with Kronos. The idea is to design applications in discrete time and to show their correctness with respect to implementations modeled in continuous time. The semantics of Quartz programs have been defined using timed Kripke structures for the verification of timing properties [24] and more recently extended to treat hybrid systems [4]. Abstract interpretation techniques have been proposed to verify safety properties of synchronous models involving multiform timing characteristics linked by linear relations [18]. We focus on a different problem: the symbolic simulation of real-time models involving nondeterminism. Instead of adding real-time assumptions to a discrete-time program, we start from a continuous-time model and show how an execution can be discretized.

8 Conclusion

We combine existing techniques and data structures for Timed Safety Automata with typing and compilation techniques for synchronous languages to develop a novel programming language where discrete reactive logic can be mixed with nondeterministic continuous-time features. In particular, we present an extension of Lustre and a specialization of Zélus for modeling real-time reactive systems, propose a symbolic simulation scheme based on ‘sweeping’, and show how to implement it via source-to-source compilation. A type system ensures the correct composition of discrete-time and continuous-time elements.

Our proposal has been implemented using the Zélus compiler and a small library of DBM operations.[8] Its application to a standard academic example is described elsewhere [3, §6.7]. Other possible extensions include the use of octagons [26] to extend the constraint language with sums of timers, and the suspension of timers in inactive automaton states as in stopwatch automata [14]. Our ultimate goal is to provide developers of reactive systems with practical and precise programming languages that allow both simulation and analysis before discretization on a given platform, and efficient compilation to executable code afterward.

[8] `https://github.com/gbdrt/zsy/tree/fdl17`

References

1. Rajeev Alur. Formal verification of hybrid systems. In *International Conference on Embedded Software (EMSOFT)*, pages 273–278, Taiwan, October 2011.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. Guillaume Baudart. *A Synchronous Approach to Quasi-Periodic Systems*. PhD thesis, PSL Research University, March 2017.
4. Kerstin Bauer and Klaus Schneider. From synchronous programs to symbolic representations of hybrid systems. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 41–50, Stockholm, Sweden, April 2010. ACM Press.
5. Gerd Behrmann, Alexandre David, Kim G. Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 125–126, Riverside, California, USA, September 2006. IEEE Computing Society.
6. Johan Bengtsson. *Clocks, DBMs and states in timed systems*. PhD thesis, Uppsala University, 2002.
7. Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *Conference on Languages, Compilers, and tools for embedded systems (LCTES)*, pages 61–70, USA, April 2011.
8. Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. A hybrid synchronous language with hierarchical automata: Static typing and translation to synchronous code. In *International Conference on Embedded Software (EMSOFT)*, Taiwan, October 2011.
9. Bernard Berthomieu and Miguel Menasche. An enumerative approach for analyzing Time Petri Nets. In *World Computer Congress (IFIP)*, pages 41–46, France, September 1983.
10. V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos: A tool for verifying real-time properties of embedded systems. In *CDC*, pages 2875–2880, Orlando, Florida, USA, December 2001. IEEE.
11. Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 113–118, USA, April 2013.
12. Paul Caspi. The quasi-synchronous approach to distributed control systems. Technical Report CMA/009931, VERIMAG, Crysis Project, May 2000. *The Cooking Book*.
13. Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages (POPL)*, pages 178–188, Germany, January 1987.
14. Franck Cassez and Kim G. Larsen. The impressive power of stopwatches. In *International Conference on Concurrency Theory (CONCUR)*, pages 138–152, USA, August 2000.
15. Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 model checker. In *International Conference on Computer Aided Verification (CAV)*, pages 510–517, Canada, July 2016.
16. David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Workshop on Automatic verification methods for finite state systems (AVMFSS)*, pages 197–212, France, June 1990.
17. David Garriou. Symbolic simulation of synchronous programs. *Elecronic Notes in Theoretical Computer Science*, 65(5):11–18, 2002.
18. Nicolas Halbwachs. Delay analysis in synchronous programs. In *International Conference on Computer Aided Verification (CAV)*, pages 333–346, Greece, June 1993.
19. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):192–244, June 1994.
20. Tobias Isenberg and Heike Wehrheim. Timed automata verification via IC3 with zones. In *International Conference on Formal Methods and Software Engineering (ICFEM)*, volume 8829 of *Lecture Notes in Computer Science*, pages 203–218, November 2014.

21. Martin Jourdan, Florence Maraninchi, and Alfredo Olivero. Verifying quantitative real-time properties of synchronous programs. In *International Conference on Computer Aided Verification (CAV)*, Greece, June 1993.
22. Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. SMT-based induction methods for timed systems. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 7595 of *Lecture Notes in Computer Science*, pages 171–187, September 2012.
23. Kim G. Larsen, Paul Pettersson, and Yi Wang. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
24. George Logothetis and Klaus Schneider. Extending synchronous languages for generating abstract real-time models. In *Design, Automation, and Test in Europe (DATE)*, France, March 2002.
25. Christophe Mauras. Symbolic simulation of interpreted automata. In *International Workshop on Synchronous Programming (SYNCHRON)*, Germany, December 1996.
26. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
27. Georges Morbé, Florian Pigorsch, and Christoph Scholl. Fully symbolic model checking for timed automata. In *International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 616–632, July 2011.
28. Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal of Embedded Systems*, 2008.
29. Pascal Raymond, Yvan Roux, and Erwan Jahier. Specifying and executing reactive scenarios with Lutin. *Elecronic Notes in Theoretical Computer Science*, 203(4):19–34, 2008.
30. F.W. Vaandrager and A.L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing*, 18(4):433–458, December 2006.
31. Farn Wang. Efficient verification of timed automata with BDD-like data structures. *International Journal on Software Tools for Technology Transfer*, 6:77–97, July 2004.
32. Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1):123–133, 1997.